

合肥工业大学

操作系统实验报告

实验题目 实验 5 进程的同步

学生姓名 袁焕发

学 号 2019217769

专业班级 物联网工程 19-2

指导教师 田卫东

完成日期 2021 年 11 月 9 日

1. 实验目的和任务要求

使用 EOS 的信号量，编程解决生产者—消费者问题，理解进程同步的意义。
调试跟踪 EOS 信号量的工作过程，理解进程同步的原理。

修改 EOS 的信号量算法，使之支持等待超时唤醒功能（有限等待），加深理解进程同步的原理。

2. 实验原理

多个并发执行的进程可以同时访问的硬件资源（打印机、磁带机）和软件资源（共享内存）都是临界资源。由于进程的异步性，当它们争用临界资源时，会给系统造成混乱，所以必须互斥地对临界资源进行访问。我们把在每个进程中访问临界资源的那段代码称为临界区（Critical Section）。可以使用互斥体保证各进程互斥地进入自己的临界区，从而保证各进程互斥地访问临界资源。

EOS 内核提供了三种专门用于线程同步的内核对象：互斥（Mutex）对象、信号量（Semaphore）对象和事件（Event）对象。另外，EOS 中的进程对象和线程对象也支持同步功能。EOS 中所有支持同步功能的对象都有两种状态：signaled 和 nonsignaled，线程对处于 nonsignaled 状态的同步对象执行 Wait 操作将会被阻塞，直到同步对象的状态变为 signaled，EOS 内核为同步对象提供了统一的 Wait 操作接口函数。

3. 实验内容

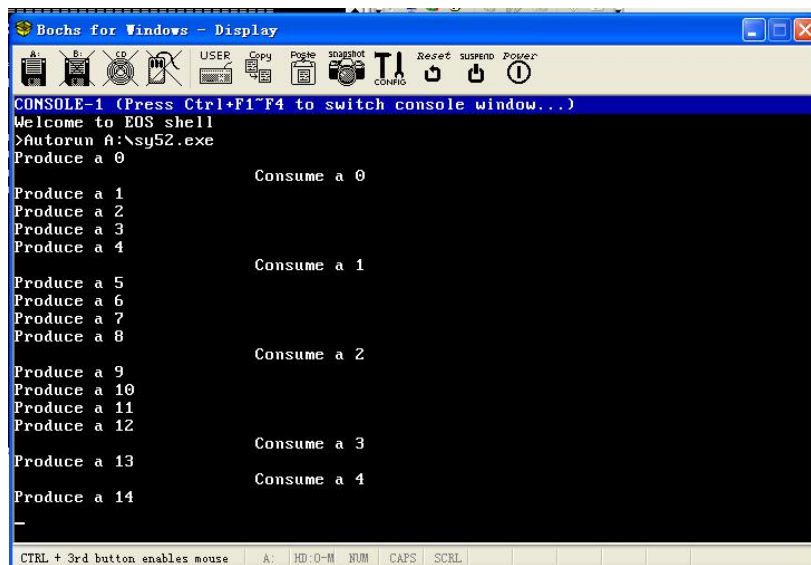
3.1. 准备实验

启动 OS Lab，新建一个 EOS Kernel 项目，生成 EOS Kernel 项目，从而在该项目文件夹中生成 SDK 文件夹。新建一个 EOS 应用程序项目，用在 EOS Kernel 项目生成的 SDK 文件夹覆盖 EOS 应用程序项目文件夹中的 SDK 文件夹。

3.2. 使用 EOS 的信号量解决生产者—消费者问题

用“学生包”本实验对应的文件夹中 pc.c 文件中的源代码，替换之前创建的 EOS 应用程序项目中 EOSApp.c 文件内的源代码，按 F7 生成修改后的 EOS 应用程序项目，按 F5 启动调试，立即激活虚拟机窗口查看生产者—消费者同步

执行的过程。



3.3. 调试 EOS 信号量的工作过程

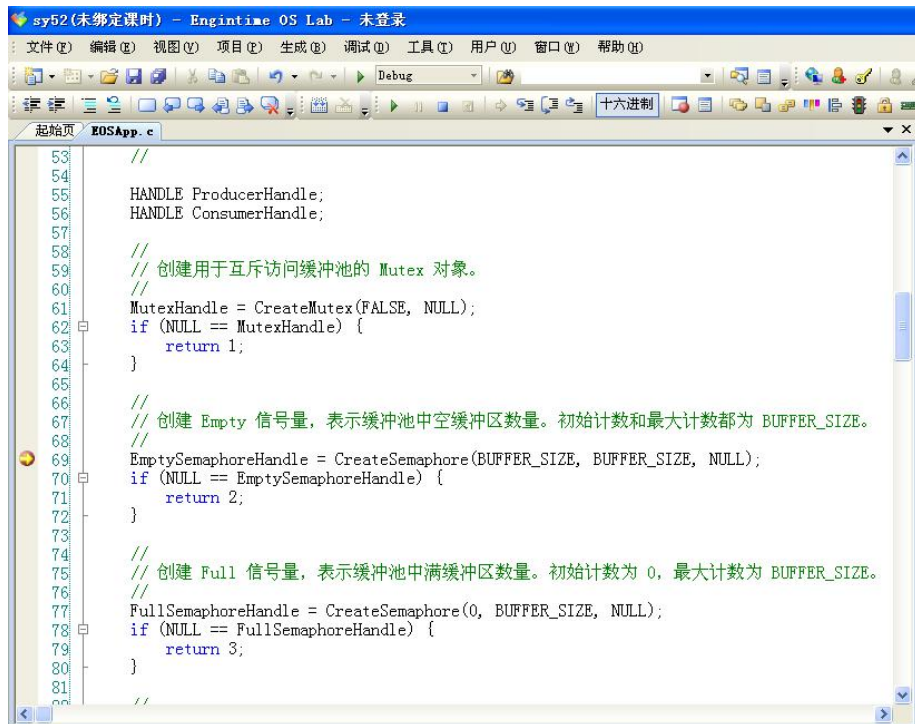
3.3.1. 创建信号量

在 main 函数中创建 Empty 信号量的代码行（第 69 行）

```
EmptySemaphoreHandle = CreateSemaphore(BUFFER_SIZE, BUFFER_SIZE, NULL);
```

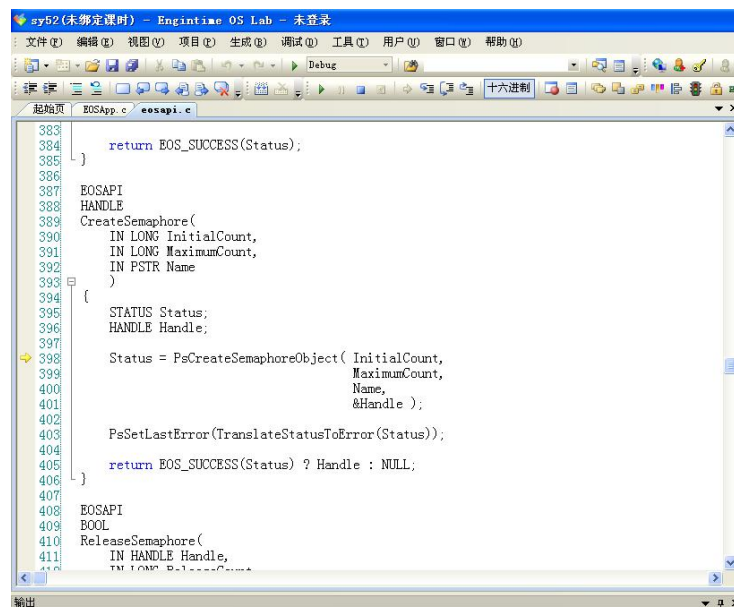
添加一个断点。按 F5 启动调试，到此断点处中断。

按 F11 调试进入 CreateSemaphore 函数。可以看到此 API 函数只是调用了 EOS 内核中的 PsCreateSemaphoreObject 函数来创建信号量对象。



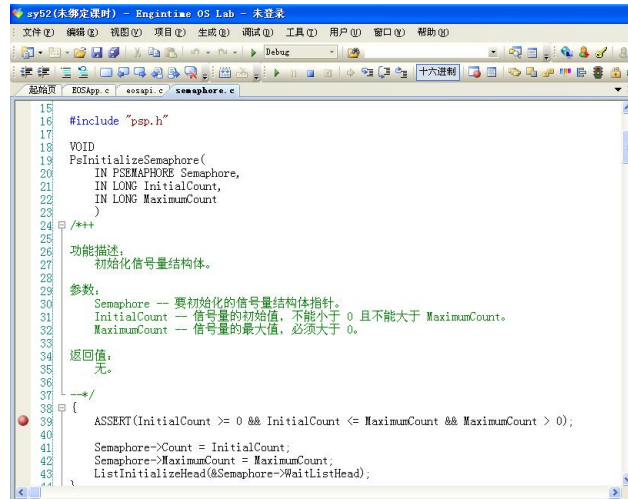
```
53 //
54
55 HANDLE ProducerHandle;
56 HANDLE ConsumerHandle;
57
58 //
59 // 创建用于互斥访问缓冲池的 Mutex 对象。
60 //
61 MutexHandle = CreateMutex(FALSE, NULL);
62 if (NULL == MutexHandle) {
63     return 1;
64 }
65
66 //
67 // 创建 Empty 信号量，表示缓冲池中空闲缓冲区数量。初始计数和最大计数都为 BUFFER_SIZE。
68 //
69 EmptySemaphoreHandle = CreateSemaphore(BUFFER_SIZE, BUFFER_SIZE, NULL);
70 if (NULL == EmptySemaphoreHandle) {
71     return 2;
72 }
73
74 //
75 // 创建 Full 信号量，表示缓冲池中已满缓冲区数量。初始计数为 0，最大计数为 BUFFER_SIZE。
76 //
77 FullSemaphoreHandle = CreateSemaphore(0, BUFFER_SIZE, NULL);
78 if (NULL == FullSemaphoreHandle) {
79     return 3;
80 }
81
82 //
```

按 F11 调试进入 CreateSemaphore 函数。按 F11 调试进入 semaphore.c 文件中的 PsCreateSemaphoreObject 函数。



```
383
384     return EOS_SUCCESS(Status);
385 }
386
387 EOSAPI
388 HANDLE
389 CreateSemaphore(
390     IN LONG InitialCount,
391     IN LONG MaximumCount,
392     IN PSTR Name
393 )
394 {
395     STATUS Status;
396     HANDLE Handle;
397
398     Status = PsCreateSemaphoreObject( InitialCount,
399                                     MaximumCount,
400                                     Name,
401                                     &Handle );
402
403     PsSetLastError(TranslateStatusToError(Status));
404
405     return EOS_SUCCESS(Status) ? Handle : NULL;
406 }
407
408 EOSAPI
409 BOOL
410 ReleaseSemaphore(
411     IN HANDLE Handle,
```

在 semaphore.c 文件的 PsInitializeSemaphore 函数的定义（第 19 行），在此函数的第一行（第 39 行）代码处添加一个断点。



按 F5 继续调试，到断点处中断。按 F10 单步调试 PsInitializeSemaphore 函数到第 44 行。

打开“调用堆栈”窗口，查看函数的调用层次。选择“调试”菜单“窗口”中的“记录型信号量”，打开“记录型信号量”窗口。在该窗口工具栏上点击“刷新”按钮，可以看到当前系统中已经有一个创建完毕的信号量。

数据源: SEMAPHORE Semaphore

源文件: ps\semaphore.c

记录型信号量

| 序号 | 信号量的整型值 (Count) | 允许最大值 (MaximumCount) | 阻塞线程链表 (Wait) |
|----|--------------------|-------------------------|------------------|
| 1 | 0xa | 0xa | NULL |

3.4. 等待信号量 (P 操作) 和释放信号量 (V 操作)

首先需要删除所有断点，在 eosapp.c 文件的 Producer 函数中，WaitForSingleObject(EmptySemaphoreHandle, INFINITE)添加一个断点，按 F5 继续调试，到断点处中断。在 semaphore.c 文件中 PsWaitForSemaphore 函数的第一行（第 68 行）添加一个断点，按 F5 继续调试，到断点处中断，按 F10 单步调试，直到完成 PsWaitForSemaphore 函数中的所有操作。刷新“记录型信号量”窗口，显示如图的内容，可以看到此次执行并没有进行等待，只是将 Empty 信号量的计数减少了 1（由 10 变为了 9）就返回了。

数据源: SEMAPHORE Semaphore
源文件: ps\semaphore.c

记录型信号量

| 序号 | 信号量的整型值 (Count) | 允许最大值 (MaximumCount) | 阻塞线程链表 (Wait) |
|----|--------------------|-------------------------|------------------|
| 1 | 0x9 | 0xa | NULL |
| 2 | 0x0 | 0xa | NULL |

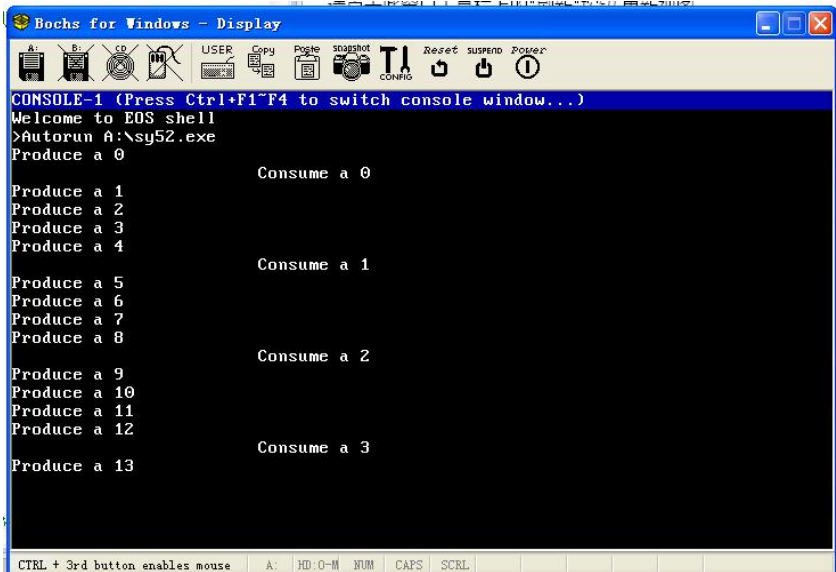
删除所有的断点，在 eosapp.c 文件的 Producer 函数中，释放 Full 信号量的代码行（第 144 行）ReleaseSemaphore(FullSemaphoreHandle, 1, NULL) 添加一个断点，按 F5 继续调试，到断点处中断，按 F11 调试进入 ReleaseSemaphore 函数，继续按 F11 调试进入 PsReleaseSemaphoreObject 函数，先使用 F10 单步调试，当黄色箭头指向第 269 行时使用 F11 单步调试，进入 PsReleaseSemaphore 函数，继续按 F10 单步调试，直到完成 PsReleaseSemaphore 函数中的所有操作。刷新“记录型信号量”窗口，可以看到此次执行没有唤醒其它线程（因为此时没有线程在 Full 信号量上被阻塞），只是将 Full 信号量的值增加了 1（由 0 变为了 1）。

数据源: SEMAPHORE Semaphore
源文件: ps\semaphore.c

记录型信号量

| 序号 | 信号量的整型值 (Count) | 允许最大值 (MaximumCount) | 阻塞线程链表 (Wait) |
|----|--------------------|-------------------------|------------------|
| 1 | 0x9 | 0xa | NULL |
| 2 | 0x1 | 0xa | NULL |

结束之前的调试，删除所有的断点，在 semaphore.c 文件中的 PsWaitForSemaphore 函数的 PspWait(&Semaphore->WaitListHead, INFINITE); 代码行（第 78 行）添加一个断点，按 F5 启动调试，并立即激活虚拟机窗口查看输出。开始时生产者、消费者都不会被信号量阻塞，同步执行一段时间后在断点处中断。



中断后，查看“调用堆栈”窗口，有 Producer 函数对应的堆栈帧，说明此次调用是从生产者线程函数进入的。



刷新“记录型信号量”窗口，查看 Empty 信号量计数（Semaphore->Count）的值为-1，所以会调用 PspWait 函数将生产者线程放入 Empty 信号量的等待队列中进行等待，使之让出处理器。。

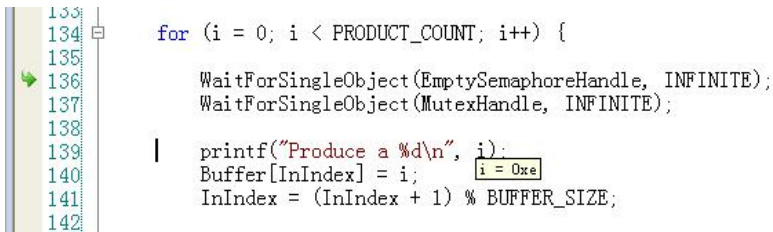
数据源: SEMAPHORE Semaphore

源文件: ps\semaphore.c

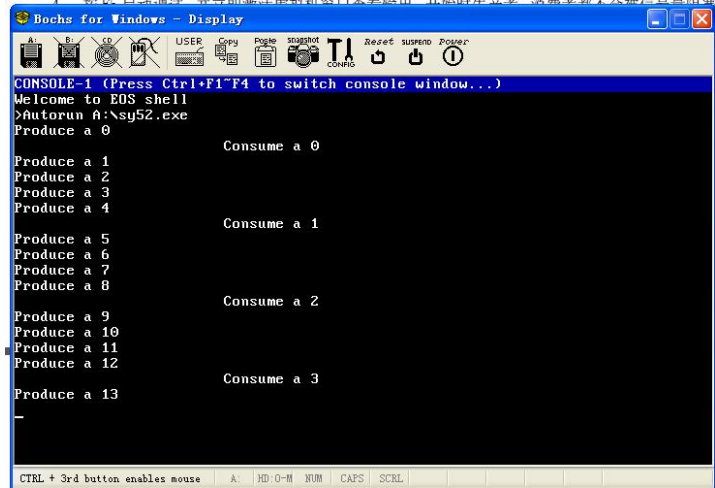
记录型信号量

| 序号 | 信号量的整型值 (Count) | 允许最大值 (MaximumCount) | 阻塞线程链表 (Wait) |
|----|--------------------|-------------------------|------------------|
| 1 | 0xffffffff | 0xa | NULL |
| 2 | 0xa | 0xa | NULL |

在“调用堆栈”窗口中双击 Producer 函数所在的堆栈帧，绿色箭头指向等待 Empty 信号量的代码行，查看 Producer 函数中变量 i 的值为 14，表示生产者线程正在尝试生产 14 号产品。



激活虚拟机窗口查看输出的结果。生产了从 0 到 13 的 14 个产品，但是只消费了从 0 到 3 的 4 个产品，所以缓冲池中的 10 个缓冲区就都被占用了，这与之前调试的结果是一致的。



删除所有断点，在 eosapp.c 文件的 Consumer 函数中，释放 Empty 信号量的代码行（第 172 行）ReleaseSemaphore(EmptySemaphoreHandle, 1, NULL); 添加一个断点。按 F5 继续调试，会在断点处中断。刷新“记录型信号量”窗口，会显示如下图所示的内容，可以看到此时生产者线程仍然阻塞在 Empty 信号量上。

数据源: SEMAPHORE Semaphore
源文件: ps\semaphore.c

| 记录型信号量 | | | |
|--------|-----------------|----------------------|---------------|
| 序号 | 信号量的整型值 (Count) | 允许最大值 (MaximumCount) | 阻塞线程链表 (Wait) |
| 1 | 0xffffffff | 0xa | |
| 2 | 0x9 | 0xa | NULL |

item

| | |
|----------|------|
| thread | next |
| TID = 30 | NULL |

查看 Consumer 函数中变量 i 的值为 4，说明已经消费了 4 号产品

```
for (i = 0; i < PRODUCT_COUNT; i++) {  
    i = 0x4  
    WaitForSingleObject(FullSemaphoreHandle, INFINITE);  
    WaitForSingleObject(MutexHandle, INFINITE);  
}
```

查看 PsReleaseSemaphore 函数中 Empty 信号量计数 (Semaphore->Count) 的值为-1，和生产者线程被阻塞时的值是一致的。

```
STATUS Status;  
BOOL IntState;  
  
IntState = KeEnableInterrupts(FALSE); // 开始原子操作，禁止中断。  
  
if (Semaphore->Count + ReleaseCount > Semaphore->MaximumCount) {  
    Semaphore->Count = 0xffffffff  
    Status = STATUS_SEMAPHORE_LIMIT_EXCEEDED;  
}
```

按 F10 单步调试 PsReleaseSemaphore 函数，直到在代码行（第 132 行）PspWakeThread(&Semaphore->WaitListHead, STATUS_SUCCESS)处中断。此时，刷新“记录型信号量”窗口，可以看到 Empty 信号量计数的值已经由-1 增加为了 0。

数据源: SEMAPHORE Semaphore
源文件: ps\semaphore.c

| 记录型信号量 | | | |
|--------|-----------------|----------------------|---------------|
| 序号 | 信号量的整型值 (Count) | 允许最大值 (MaximumCount) | 阻塞线程链表 (Wait) |
| 1 | 0x0 | 0xa | |
| 2 | 0x9 | 0xa | NULL |

item

| | |
|----------|------|
| thread | next |
| TID = 30 | NULL |

在 semaphore.c 文件中 PsWaitForSemaphore 函数的最后一行（第 83 行）代码处添加一个断点。按 F5 继续调试，在断点处中断，查看 PsWaitForSemaphore 函数中 Empty 信号量计数 (Semaphore->Count) 的值为 0，

和生产者线程被唤醒时的值是一致的。

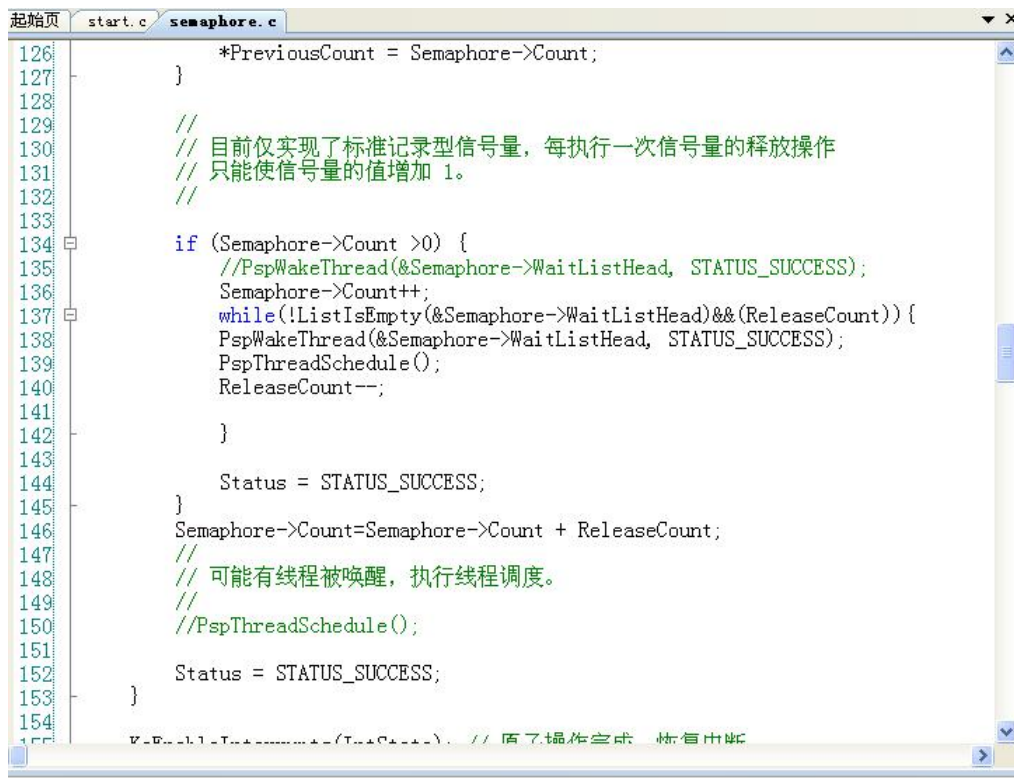
```
75 //
76 Semaphore->Count--;
77 if (Semaphore->Count < 0) {
78     PspWait(&Semaphore->Count, INFINITE);
79 }
80
81 KeEnableInterrupts(IntState); // 原子操作完成, 恢复中断。
82
83 return STATUS_SUCCESS;
```

在“调用堆栈”窗口中可以看到是由 Producer 函数进入的。激活 Producer 函数的堆栈帧, 查看 Producer 函数中变量 i 的值为 14, 表明之前被阻塞的、正在尝试生产 14 号产品的生产者线程已经从 PspWait 函数返回并继续执行了。

```
129 ULONG Producer(PVOID Param)
130 {
131     int i;
132     int InIndex = 0;
133
134     for (i = 0; i < PRODUCT_COUNT; i++) {
135         i = 0xa;
136         WaitForSingleObject(EasySemaphoreHandle, INFINITE);
137         WaitForSingleObject(MutexHandle, INFINITE);
138     }
```

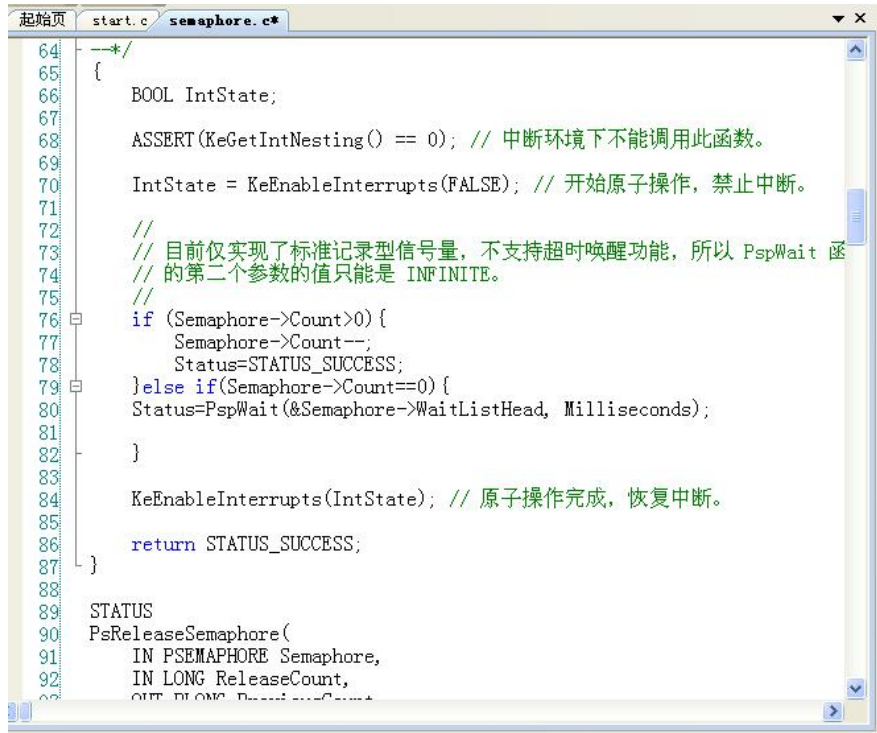
3.5. 修改 EOS 的信号量算法

修改 PsWaitForSemaphore 函数时要注意, 对于支持等待超时唤醒功能的信号量, 其计数值只能是大于等于 0。当计数值大于 0 时, 表示信号量为 signaled 状态; 当计数值等于 0 时, 表示信号量为 nonsignaled 状态。所以 PsWaitForSemaphore 函数中原有的代码段应被修改为: 先用计数值和 0 比较, 当计数值大于 0 时, 将计数值减 1 后直接返回成功; 当计数值等于 0 时, 调用 PspWait 函数阻塞线程的执行(将参数 Milliseconds 做为 PspWait 函数的第二个参数, 并使用 PspWait 函数的返回值做为返回值)。在函数开始定义一个 STATUS 类型的变量, 用来保存不同情况下的返回值, 并在函数最后返回此变量的值。



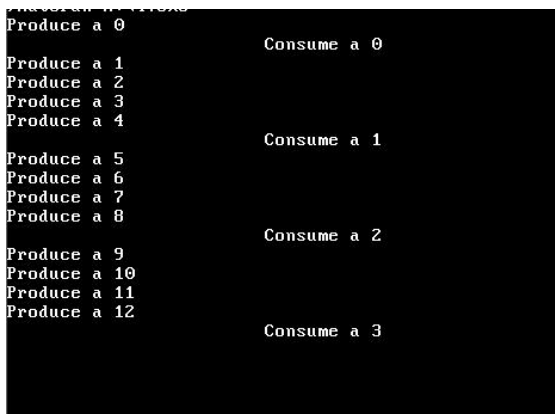
```
126     *PreviousCount = Semaphore->Count;
127 }
128
129 //
130 // 目前仅实现了标准记录型信号量, 每执行一次信号量的释放操作
131 // 只能使信号量的值增加 1。
132 //
133
134 if (Semaphore->Count > 0) {
135     //PspWakeThread(&Semaphore->WaitListHead, STATUS_SUCCESS);
136     Semaphore->Count++;
137     while(!ListIsEmpty(&Semaphore->WaitListHead) && (ReleaseCount)) {
138         PspWakeThread(&Semaphore->WaitListHead, STATUS_SUCCESS);
139         PspThreadSchedule();
140         ReleaseCount--;
141     }
142
143     Status = STATUS_SUCCESS;
144 }
145
146 Semaphore->Count = Semaphore->Count + ReleaseCount;
147 //
148 // 可能有线程被唤醒, 执行线程调度。
149 //
150 //PspThreadSchedule();
151
152 Status = STATUS_SUCCESS;
153 }
154
155 KeEnableInterrupts(IntState); // 原子操作完成, 恢复中断。
```

修改 PsReleaseSemaphore 函数时要注意：编写一个使用 ReleaseCount 做为计数器的循环体，来替换 PsReleaseSemaphore 函数中原有的代码，在循环体中完成下面的工作：如果被阻塞的线程数量大于等于 ReleaseCount，则循环结束后，有 ReleaseCount 个线程会被唤醒，而且信号量计数的值仍然为 0；如果被阻塞的线程数量（可以为 0）小于 ReleaseCount，则循环结束后，所有被阻塞的线程都会被唤醒，并且信号量的计数值=ReleaseCount-之前被阻塞线程的数量+之前信号量的计数值。



```
64  --*/
65  {
66      BOOL IntState;
67
68      ASSERT(KeGetIntNesting() == 0); // 中断环境下不能调用此函数。
69
70      IntState = KeEnableInterrupts(FALSE); // 开始原子操作，禁止中断。
71
72      //
73      // 目前仅实现了标准记录型信号量，不支持超时唤醒功能，所以 PspWait 函
74      // 的第二个参数的值只能是 INFINITE。
75      //
76      if (Semaphore->Count>0) {
77          Semaphore->Count--;
78          Status=STATUS_SUCCESS;
79      } else if (Semaphore->Count==0) {
80          Status=PspWait(&Semaphore->WaitListHead, Milliseconds);
81      }
82
83
84      KeEnableInterrupts(IntState); // 原子操作完成，恢复中断。
85
86      return STATUS_SUCCESS;
87  }
88
89  STATUS
90  PsReleaseSemaphore(
91      IN PSEMAPHORE Semaphore,
92      IN LONG ReleaseCount,
93      OUT PLONG PreviousCount
```

使用修改完毕的 EOS Kernel 项目生成完全版本的 SDK 文件夹，并覆盖之前的生产者—消费者应用程序项目的 SDK 文件夹。按 F5 调试执行原有的生产—消费者应用程序项目。



```
Produce a 0
Consume a 0
Produce a 1
Produce a 2
Produce a 3
Produce a 4
Consume a 1
Produce a 5
Produce a 6
Produce a 7
Produce a 8
Consume a 2
Produce a 9
Produce a 10
Produce a 11
Produce a 12
Consume a 3
```

将 WaitForSingleObject(EmptySemaphoreHandle, INFINITE);
替换为 while(WAIT_TIMEOUT == WaitForSingleObject(EmptySemaphoreHandle,
300)){ printf("Producer wait for empty semaphore timeout\n"); }
将 Consumer 函数中等待 Full 信号量的代码行
WaitForSingleObject(FullSemaphoreHandle, INFINITE);
替换为 while(WAIT_TIMEOUT == WaitForSingleObject(FullSemaphoreHandle,

```
300)) { printf("Consumer wait for full semaphore timeout\n"); }
```

启动调试新的生产者-消费者项目，查看在虚拟机中输出的结果，验证信号量超时等待功能是否能够正常执行。

```
CONSOLE-1 (Press Ctrl+F1/F8 to switch console window...)
Welcome to EOS shell
>Autorun A:\EOSApp.exe
Produce a 0
Produce a 1
Produce a 2
Produce a 3
Produce a 4
Produce a 5
Produce a 6
Produce a 7
Produce a 8
Produce a 9
Produce a 10
Produce a 11
Produce a 12
Produce a 13
Produce a 14
Consume a 0
Consume a 1
Consume a 2
Consume a 3
Consume a 4
```

如果超时等待功能已经能够正常执行，可以考虑将消费者线程修改为一次消费两个产品，来测试 `ReleaseCount` 参数是否能够正常使用，使用实验文件夹中 `NewConsumer.c` 文件中的 `Consumer` 函数替换原有的 `Consumer` 函数。

```
Produce a 3
Produce a 4
Produce a 5
Produce a 6
Produce a 7
Produce a 8
Produce a 9
Produce a 10
Produce a 11
Produce a 12
Produce a 13
Produce a 14
Produce a 15
Produce a 16
Produce a 17
Consume a 2
Consume a 3
Consume a 4
Consume a 5
Consume a 6
Consume a 7
Consume a 8
Consume a 9
```

4. 实验的思考与问题分析

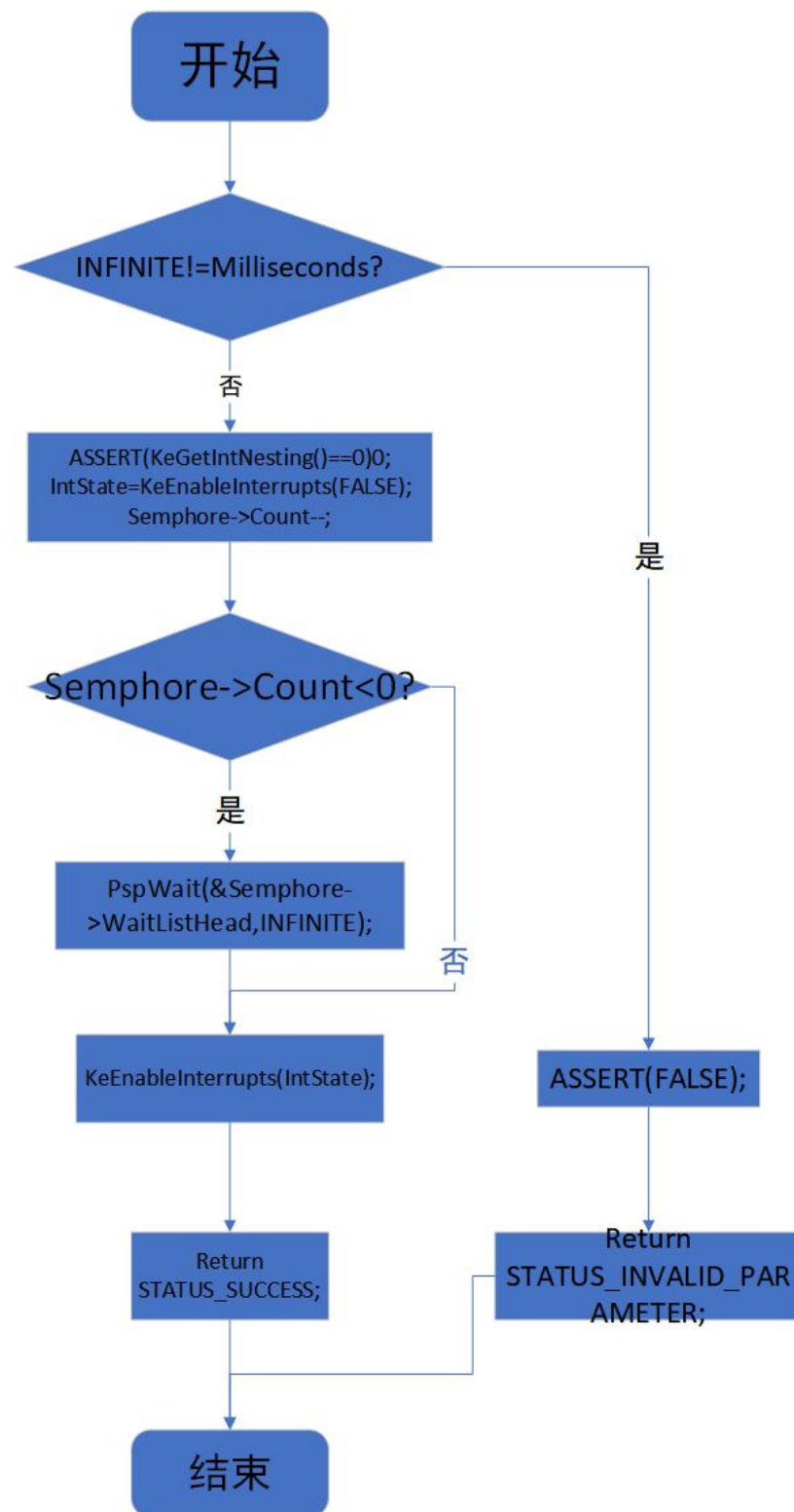
4.1. 思考 在 `ps/semaphore.c` 文件内的 `PsWaitForSemaphore` 和 `PsReleaseSemaphore` 函数中，为什么要使用原子操作？可以参考本书第 2 章中的第 2.6 节。

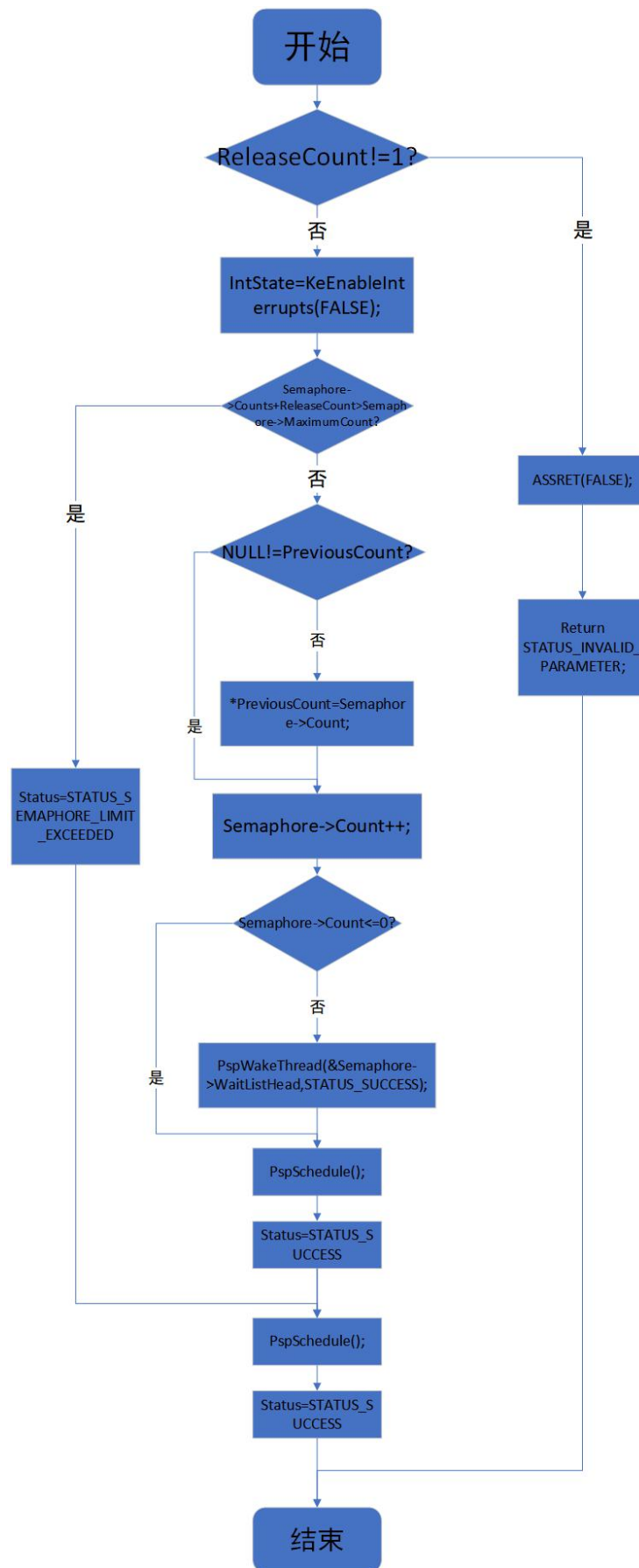
答：

在执行释放信号量和等待信号量时，是不允许 CPU 响应外部中断的，否则会产生不可预料的结果。

4.2. 绘制 ps/semaphore.c 文件内 PsWaitForSemaphore 和 PsReleaseSemaphore 函数的流程图。

答：





- 4.3. 根据本实验 3.3.2 节中设置断点和调试的方法，练习调试消费者线程在消费第一个产品时，等待 Full 信号量和释放 Empty 信号量的过程。注意信号量计数是如何变化的。

答：此题为实验练习调试。

- 4.4. 根据本实验 3.3.2 节中设置断点和调试的方法，自己设计一个类似的调试方案来验证消费者线程在消费 24 号产品时会被阻塞，直到生产者线程生产了 24 号产品后，消费者线程才被唤醒并继续执行的过程。

答：

删除之前所有的断点。在 Consumer 函数中等待 Full 信号量的代码行（第 173 行）WaitForSingleObject(FullSemaphoreHandle, INFINITE)；添加一个断点。在“断点”窗口中此断点的名称上点击右键。在弹出的快捷菜单中选择“条件”。在“断点条件”对话框的表达式编辑框中，输入表达式“i == 24”。点击“断点条件”对话框中的“确定”按钮。按 F5 启动调试。只有当消费者线程尝试消费 24 号产品时才会在该条件断点处中断。

- 4.5. 创建多个生产者线程和多个消费者线程进行同步，注意临界资源也会发生变化。然后，添加断点，调试程序，在“记录型信号量”窗口中查看信号量的变化情况和阻塞在信号量上的线程信息，同时，在“互斥信号量”窗口中查看互斥信号量的变化情况和阻塞在互斥信号量上的线程信息。

答：已在实验中调试练习。

5. 总结和感想体会

首先我明白了生产者—消费者问题的模型，生产者生产者线程生产物品，然后将物品放置在一个空缓冲区中供消费者线程消费。消费者线程从缓冲区中获得物品，然后释放缓冲区。当生产者线程生产物品时，如果没有空缓冲区可用，那么生产者线程必须等待消费者线程释放出一个空缓冲区。当消费者线程消费物品时，如果没有满的缓冲区，那么消费者线程将被阻塞，直到新的物品被生产出来。然后通过调试和使用 EOS 的 Mutex、Empty 信号量和 Full 信号量，学会了处理生产者—消费者问题，理解了进程同步的意义。在调试跟踪信号量的过程中，理解了进程同步的原理。根据指导学会了修改信号量算法，完成等待超时唤醒功能，加深了对进程同步原理的理解。在断点调试中，明白了 EOS 内核提供的 Semaphore 对象是典型的记录型信号量，知道了该函数中各个参数的意义。